HW8

1. Dynamically typed languages utilize duck typing and so they don't require interfaces. Duck typing allows for the object method and properties to determine the valid semantics where as having an interface would rely on some sort of class to inherit properties from which aren't necessary in a dynamically typed language whose properties/types is classified at runtime.
2. A time when I would prefer interface inheritance over traditional subclass inheritance would be when I want to guaranteee that a class adheres to a certain specification, but when the implementation of that class isn't as important.

For example: ``` interface Drawable { void draw(); }

```
class Circle implements Drawable {
        // Implementation of draw for Circle
        public void draw() {
            // Draw a circle
        }
    }

class Rectangle implements Drawable {
        // Implementation of draw for Rectangle
        public void draw() {
            // Draw a rectangle
        }
    }```
```

If I had the following implementation in Java, I would utilize interface inheritance for the purpose of each class having its own way of defining the drawing function but they all adhere to the overarching Drawable interface.

A time when I would prefer traditional inheritance would be when I want to ensure that there is a clear relationship between the classes and I want to utilize code from the parent class in the child class.

For example:

```
    class Vehicle {
        protected int speed;
        protected int direction;

        public void accelerate(int amount) {
            // Increase speed
        }

        public void turn(int degrees) {
            // Change direction
        }
    }

    class Car extends Vehicle {
        private int fuelLevel;
```

```
        public void refuel(int amount) {
            // Increase fuel level
        }
    }

    class Bicycle extends Vehicle {
        private boolean isBellRinging;

        public void ringBell() {
            // Ring the bell
        }
    }```
In this example, I found that using traditional inheritance was more optimal since it would allow for the subclasses to inherit from the parent classes when i didn't want to overwrite a function.



3.
   a.
   - Interface A doesn't have supertypes
   - Interface B, C has supertype A
   - Class D has supertypes B, C, A
   - Class E has supertypes C, A
   - Class F has supertypes D, B, C, A
   - Class G has supertypes B, A

    b.
    foo(B b) is able to accept objects of class D, F, G because they use the interface B either directl or indirectly via class D.

    c.
    Bletch is not able to call bar because bar requires an object of type C and the object type given is of type A. object A is not a subtype of C and so the bar function is not able to accept C as an input and thus, it will not work as intended



4.
   Inheritance is when one defines a new class based on the contents of an existing class. So the new class "inherits" the properties of the existing class. Through inheritance we are able to pass down the methods and fields of the members and don't need to rewrite the code for the subclass.

   Subtype polymorphism is when a subclass is able to be substituted in any situation, where the superclass is expected. In class, we saw that the Nerd class could be used where a Person class Object was expected. This enables flexibility and also code reusability but in the opposite manner of inheritance.

   Dynamic Dispatch is a mechanims that occurs when a call to an overriden method is resolved at runtime instead of compile time. We use dynamic dispatch to enable polymorphism. During runtime, we will actually execute the method being called on an object and it is determined at runtime. The type of the reference isn't utilized to
```

call the method of the object. Thus, dynamic dispatch allows us to revise the implementation of a method that is already supported by its superclass.

5.

We can't use subtype polymorphism in dynamically typed languages because of the duck typing. Duck typing will already determine which method to utlize based on the type of the object determined at runtime and so the type/class of an object is less important than the methods/members of the object itself. Duck typing essentially only cares about the members/fields/methods not the actual inheritance involved that is characteristic of Subtype polymorphism.

We are most definitely able to use dynamic dispatch in dynamically typed languages because it determines the overridden method at runtime. This mechansim is likely quite commonly used in dynamically typed languages since the actual method being executed is determined at runtime. An example of where this would be useful is in the code below:

```class Animal:
 def speak(self):
     pass

 class Dog(Animal):
     def speak(self):
         return "Woof!"

 class Cat(Animal):
     def speak(self):
         return "Meow!"

 def make_animal_speak(animal):
     print(animal.speak())

 dog = Dog()
 cat = Cat()

 make_animal_speak(dog)  # prints "Woof!"
 make_animal_speak(cat)  # prints "Meow!"```

Here we can take a method speak and call it on an object like animal.speak() and then the method that gets executed will be determined at runtime based on the type of the animal object.
```